COMP 6360 Wireless and Mobile Networks
Project 1 Report

Team 2*

Mar. 7, 2012

1 Architecture

The simulator consists of three threads and multiple classes. As shown in Figure 1, they work collaboratively to simulate the EEBL application. Section 2 discusses details of each part of the system.

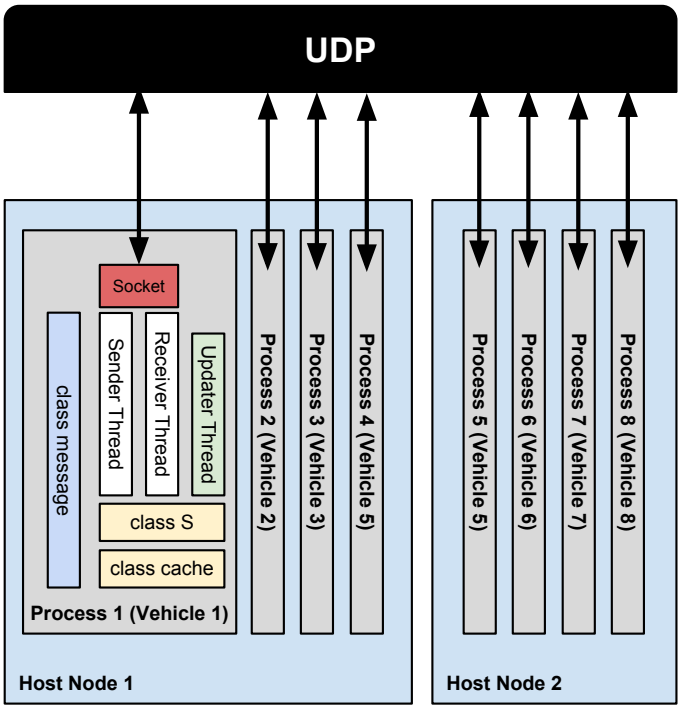

Figure 1: Architecture

2 Design

2.1 Message modeling and serialization

Involved classes: `message`, `vehicle`

`message` class is in charge of modeling and manipulating messages. It has three main responsibilities:

2.1.1 Carrying Data

We use a same message structure for both beacons and EEBL messages. It takes 46 bytes in amount. Table 1 shows information carried by `message` class.

Since vehicle IDs and packet IDs have same size (4 bytes), we have to suppose that packet IDs are unique in a single vehicle, however, different vehicles may have identical packet IDs. Therefore, vehicle ID and packet ID combined together can be used as unique packet identifier in the whole network.

Data	Size	Data	Size
Message Type	1	Vehicle Heading	2
Packet ID	4	Vehicle Size	6
Originator ID	4	Vehicle Speed	2
Sender ID	4	Vehicle Acceleration	2
Time-to-live	1	GPS Coordinate	12
Timestamp	8		

Table 1: Message Format

2.1.2 Build Messages

In addition to carrying data, `message` class is also capable of creating three kind of messages: BEACON, EEBL, and Rebroadcasted EEBL. It automatically accomplishes some of message operations that are not dependent on upper layer code. For example, it updates TTL when rebroadcasting EEBL message; and it retrieves static vehicle informations such as vehicle ID/heading. It is designed to make upper-layer code provide as little extra data as possible when creating messages.

2.1.3 Serialization and Deserialization

Finally, `message` class also provides interface to manipulate byte stream which is used by sockets to transmit over UDP. Two methods,

```
static message * message::from_bytes(
        const unsigned char data[]);
void message::to_bytes(
        unsigned char buffer[]) const;
```

, are implemented to transform between byte array and message object. By encapsulation lower-layer functions like `memcpy`, it enables upper-layer code to only deal with structured data.

2.2 Concurrent Data Share

In some situations, multiple concurrent threads need to share data and synchronize operations. For example, we have an application-layer thread(Section 2.5.3) to update vehicle state such as speed and acceleration, and (in the future) to determine whether collision will take place. This thread needs to share data with sending thread and receiving thread, which deal with socket communication.

*Jiao Yu [jzy0012], Song Gao [szg0031], Xinyu Que [xzq0003] (ordered by name)

To avoid corrupted data operation, `S` class, a thread-safe singleton class, is implemented. `S` ensures that only one instance exists in a process. It stores data that needs to be available globally to multiple threads, and use locking mechanism to synchronize concurrency operations in different threads. We choose `std::mutex` and `std::lock_guard` from `C++0x` to implement locks. Lock guarded data includes *GPS coordinate, speed, acceleration, EEBL message queue*. In addition to these shared state data, a thread-safe cache(see Section 2.3 for details) is also accessible via `S` class.

`S` class is initialized at the beginning of `main` function. It takes configuration file path and node number as arguments, and pass them to `configuration` class constructor(see Section 2.8 for details) during initialization phase.

2.3 Fast Cache For Messages and EEBL Identifiers

`cache` class implements a simple cache that stores recently received messages(including both BEACON and EEBL) and unique identifiers of recently received EEBL. The former one indicates state of other vehicles, which is used to calculate distance-based rebroadcasting probability as well as predicting collision, etc., while the later one is used to determine a certain rebroadcasted EEBL message has already been received before. To make it thread-safe, a locking mechanism similar to `S` class is used.

To achieve good performance for both data retrieval/replacement and inserted-time-based removal, `cache` class uses hash tables (`std::unordered_map/std::unordered_set`) to store the actual data, and binary search trees (`std::multimap`) to index the stored data by timestamp. In this way, data is accessible from hash tables with high performance, and since indexes are sorted on timestamp, removing outdated items can be accomplished via index very fast. Currently, we set alive time for messages and EEBL identifiers to be 8 seconds and 4 seconds.

2.4 `configuration` class Design

Reading data from the configuration file is encapsulated by the configuration class. The configuration file in our project has the format of Node 1 tux175 10120 links 2 GPS 120 225, which basically means Node 1 is simulated by tux175 with port number 10120,and connected to Node 2, also its GPS coordinate is (120, 225). The Configuration class builds three maps (implemented by C++ STL) for recording the $< tux\#, port\# >$ pair, linked nodes number as a vector, and GPS coordinate as a struct with keys node number. It provides three services to its clients, which returns the pointers to the three maps.

2.5 Threads Implementation

We split the program into 3 threads,

```
void * sender_main(void * context)
void * recver_main(void * context)
void * updater_main(void * context)
```

, to do the job of sending packets, receiving packets, and updating acceleration/deceleration and speed of vehicles respectively . In the following subsections, more details of each thread are discussed.

2.5.1 Sender Thread

First of all, the sender thread should know which nodes are within its communication range, and get their tux machine name and port numbers from the configuration file. Then a UDP socket is created, and the socket addresses of its neighboring nodes are converted from machine names and ports. There are 3 types of packets need to be sent, EEBL originated by the node itself, EEBL to be rebroadcasted, and beacon packets. The node will continuously check whether its deceleration is greater than 1m/s, which necessitates generating and sending original EEBL signal. Here we use a third threadthe updater thread, to randomly pick a acceleration of -2m/s with p = 0.2, or 2m/s with p = 0.8. For rebroadcast EEBL packets, the sender thread will check a global queue, which contains the EEBL packets that the node decides to rebroadcast according to a probability model. The queue will be filled by receiver thread, and the probability model will be discussed further in the receiver thread section. We set the priority of original EEBL packets and rebroadcast EEBL packets to be the same, which precedes beacon packets.

2.5.2 Receiver Thread

The receiver thread is responsible for receiving packets and making the decision of whether to rebroadcast a new EEBL packet. To do this job, the receiver thread first needs to get its own tux machine name and port number from the configuration file and convert it to socket address. Then it will create a UDP socket and bind that socket to its own socket address. Upon receiving a packet, the thread is going to first check the type of that packet, EEBL or beacon. If it is a EEBL packet, the receiver needs to know whether it is old or new by looking up the cache. Old packets will just be ignored and deleted, while new EEBL packets will be processed by the rebroadcast protocol. In our project, we use a probability model for rebroadcasting shown in Eq. 1.

$$P = (\frac{distance}{communicationrange})^3 \quad (1)$$

After calculating the rebroadcasting probability for a new EEBL packet, the receiver would generate a uniformly distributed random number between 0 and 1. If the random number is less than P, then this packet is going to be rebroadcasted, and will be put in the queue for the sender thread to pull rebroadcasting packets out. Also the new EEBL packets would be put in cache no matter it is going to be rebroadcasted or not. If a beacon message is received, then it is simply put in cache. Our cache is designed in such a way that it can handle both EEBL and beacon packets, maintain an index of them based on in-cache timestamp, and delete expired packets automatically.

2.5.3 Updater Thread

The updater thread is responsible for simulating the change in a vehicles acceleration and speed. For the purpose of simplicity, we set the nodes acceleration to only take two values, -2m/s or 2m/s, with probability 0.2 and 0.8 respectively, with an updating frequency of 10Hz. Speed will be updated according to the random acceleration.

2.6 Main Function

Main function function contains threads, namely send thread, receive thread and update thread. Send thread sends beacon message with a fixed frequency and receive thread receives beacon messages. Once receive thread receives EEBL messages, it will inform sender to broadcast EEBL messages. Update thread is to simulate the break of vehicle with a certain probability.

2.7 Makefile

Makefile stores compiler and linker options and expresses all the interrelationships among source files. (A source code file needs specific include files, an executable file requires certain object modules and libraries, and so forth.) Make program reads the makefile and then invokes the compiler, assembler, resource compiler, and linker to produce the final output, which is generally an executable file. The make program uses built-in inference rules that tell it, for example, to invoke the compiler to generate an OBJ file from a specified CPP file.

2.8 Launch script

A launch script is written to automatically launch the programs across many host machines, which provides a very easy and efficient way to start the program and analyze the results. In our program, the bash script first reads the `conf` file and gets the nodes information. Then it sequentially log in to host machines and launch the programs.

References

[1] Michele Segata and Renato Lo Cigno. 2011. *Emergency braking: a study of network and application performance.* In Proceedings of the Eighth ACM international workshop on Vehicular internetworking (VANET '11). ACM, New York, NY, USA, 1-10. DOI=10.1145/2030698.2030700 http://doi.acm.org/10.1145/2030698.2030700

[2] O. Tonguz, N. Wisitpongphan, J. Parikh, F. Bai, P. Mudalige, and V. Sadekar. *On the Broadcast Storm Problem in Ad hoc Wireless Networks.* In Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International Conference on, pages 111, Oct. 2006.

[3] Anthony Williams. *Simpler Multithreading in C++0x.* `http://www.devx.com/SpecialReports/Article/38883/1954`